Die mehrfädige Prozessorarchitektur Rhamma

Winfried Grünewald, Theo Ungerer

Institut für Rechnerentwurf und Fehlertoleranz
Universität Karlsruhe
76128 Karlsruhe

{gruenewald, ungerer}@informatik.uni-karlsruhe.de

Die mehrfädige (multithreaded) Prozessorarchitektur Rhamma überbrückt Wartezeiten, die von Lade-, Speicher- und Sprungbefehlen hervorgerufen werden, durch einen schnellen Kontextwechsel. Eine spezielle Befehlscodierung in Verbindung mit einer Prädecodierung während der Befehlholephase senkt den Kontextwechselaufwand auf maximal einen Takt. Ein neuartiger Kontextwechselpuffer verringert den Kontextwechselaufwand auf null Takte. Durch diese Techniken erhöht sich bei konkurrenter Ausführung mehrerer Kontrollfäden (Threads oder Prozesse) der Durchsatz des mehrfädigen Prozessors gegenüber demjenigen eines konventionellen RISC–Prozessors. Die verwendete Block–Interleaving–Technik ermöglicht, einzelne Kontrollfäden auch bei niedriger Last mit der Geschwindigkeit von konventionellen RISC–Prozessoren auszuführen. Die vorliegende Arbeit stellt Ergebnisse einer VHDL–basierten Synthese des Prozessors vor.

1. Einleitung

Der Zugriff auf den Hauptspeicher oder einen entfernten Speicher in einem Multiprozessorsystem verursacht Wartezeiten. Beim Speicherzugriff können diese Wartezeiten durch eine Hierarchie von Cache–Speichern im Mittel stark verringert werden. Weitere Verbesserungen können durch das Vorabladen von Daten per Software oder per Hardware erzielt werden. Die Zieladressen von Sprüngen können oft erst in einer späten Pipeline–Stufe berechnet werden. Diese Wartezeiten werden heute üblicherweise durch eine spekulative Sprungtechnik vermieden, wobei jedoch im Falle einer falschen Vorhersage der Sprung revidiert werden muß und damit einige Prozessortakte verbraucht werden.

Wartezeiten können auch durch einen Kontextwechsel überbrückt werden. Dies ist aber bei heutigen Mikroprozessoren wegen mangelnder Hardware–Unterstützung nicht möglich.

Das wesentliche Merkmal für mehrfädige Prozessoren ist ein schneller Kontextwechsel, der durch das Vorhandensein mehrerer Registersätze auf dem Prozessor–Chip wesentlich unterstützt wird. Man unterscheidet folgende Ansätze für mehrfädige Prozessoren:

- Die *Cycle–by–Cycle–Interleaving–Technik*: Mit jedem Prozessortakt wird ein Befehl eines anderen Kontrollfadens in die Prozessor–Pipeline eingefüttert. Im Regelfall wird ein Befehl desselben Kontrollfadens erst in die Prozessor–Pipeline eingespeist, nachdem der Vorgängerbefehl die Pipeline vollständig durchlaufen hat. Deshalb treten zwischen den Pipeline–Stufen keine von Daten- oder Kontrollabhängigkeiten ausgelöste Pipeline–Konflikte auf, und die Pipeline kann sehr einfach realisiert werden. Das bei den Rechnern HEP [1], MASA [2], Horizon [3], Tera [4] und SB–PRAM/HPP [5] eingesetzte Verfahren hat allerdings den Nachteil einer geringen Leistung, falls nur wenige Kontrollfäden als Last zur Verfügung stehen. Die Interleaving–Technik [6] verbindet die Cycle–by–Cycle–Interleaving–Technik mit einer vollausgebauten RISC–Pipeline und der Anwendung von Cache–Speichern. Dies erlaubt es, sowohl einen schnellen Kontextwechsel auszuführen als auch einen einzelnen Kontrollfaden mit der Effizienz eines RISC–Prozessors abzuarbeiten.

- Die *Block–Interleaving–Technik* [7]: Die Befehle eines Kontrollfadens werden solange aufeinanderfolgend ausgeführt, bis ein Ereignis eintritt, das zu Wartezeiten führt. Dann wird ein Kontextwechsel durchgeführt. Dieses Ereignis ist beim Sparcle–Prozessor [8] ein Cache–Fehlzugriff oder eine fehlgeschlagene Synchronisation. Der Nachteil dieser *Switch–on–Cache–Miss–Technik* ist, daß solche Cache–Fehlzugriffe erst spät in der Pipeline erkannt werden und nachfolgende bereits in der Pipeline vorhandene Befehle nicht verwendet werden können. Daraus sowie aus der teilweisen Software–Implementierung des Kontextwechsels resultiert ein Kontextwechselaufwand von 14 Takten beim Sparcle–Prozessor. Der MSparc–Prozessor [9] besitzt einen ähnlichen Aufbau wie der Sparcle, verlegt jedoch den Kontextwechsel vollständig in Hardware und kann den Kontext in einem Prozessortakt nach Auftreten des Ereignisses wechseln. Die *Nano–Threading–Technik* [10] ist eine spezielle Form der Block–Interleaving–Technik. Es wird nur ein Registersatz verwendet, den sich mehrere Kontrollfäden teilen.

- Die *Multithreaded–Superskalar–* [11] oder *Simultaneous–Multithreading–Technik* [12]: Die Ausführungseinheiten eines superskalaren Prozessors werden simultan aus mehreren Befehlsströmen bestückt.

Bei der *Decoupled Architecture* [13] wird der sequentielle Befehlsstrom vom Compiler in zwei spezialisierte Befehlssequenzen zerteilt. Die eine Sequenz bearbeitet ausschließlich Rechenbefehle, die andere Lade-/Speicherbefehle. Sprungbefehle können in beiden Befehlssequenzen verwendet werden. Zwei durch FIFO–Puffer verbundene Einheiten arbeiten zeitversetzt am *gleichen* Kontrollfaden.

Entwurfsziel des Rhamma–Prozessors ist es, einen Universalprozessor zu entwickeln, der Wartezeiten durch einen schnellen Kontextwechsel überbrückt. Dabei wird eine Kombination aus Block–Interleaving–Technik und Decoupled Architecture untersucht, die den Kontextwechsel im Befehlsstrom codiert. Lade-/Speicher- und Ausführungseinheit sind entkoppelt und arbeiten gleichzeitig an *verschiedenen* Kontrollfäden. Dies wird durch das Vorhandensein mehrerer Registersätze auf dem Prozessor–Chip ermöglicht. Anders als bei ähnlichen mehrfädigen Prozessoren mit Block–Interleaving–Technik wird bereits in der Befehlholephase der Pipeline erkannt, ob ein Kontextwechsel vorgenommen werden muß (*Switch–on–Load–Strategie*). Dadurch kann der Aufwand für einen Kontextwechsel gegenüber Prozessoren mit Switch–

on–Cache–Miss–Technik reduziert werden. Allerdings werden Kontextwechsel häufiger durchgeführt.

Der grundlegende Entwurf des Rhamma–Prozessors wurde bereits in [14] und [15] vorgestellt. Es wurde ein ereignisorientierter Software–Simulator entwickelt, der sowohl das Verhalten eines Einprozessorsystems wie auch das eines speichergekoppelten Multiprozessorsystems auf Register–Transfer–Ebene nachbildet. Grundlage für die Ausführungseinheit bildet ein konventioneller RISC–Prozessor (eine Variante des DLX–Prozessors der Universität Stanford [16]), dessen Befehlssatz um Kontext–Management–Befehle erweitert wurde.

Software–Simulationen eines speichergekoppelten Multiprozessorsystems mit dem Rhamma–Prozessor als Knoten zeigten, daß die präsentierte mehrfädige Prozessortechnik geeignet ist, bei genügender Last lange Speicherzugriffszeiten durch den schnellen Kontextwechsel zu überbrücken und dadurch Sekundär–Cache–Speicher zu ersetzen [17]. Verwendet man den Rhamma–Prozessor in einer Workstation, so zeigten Sofwaresimulationen seine Fähigkeit, auch kurze Wartezeiten zu überbrücken, sofern die Kontextwechselzeit sehr gering ist (in der Größenordnung von ein bis vier Prozessortakten), wobei Registersätze für drei bis vier Kontrollfäden ausreichen ([15], [18]).

Mittels Software–Simulationen des Rhamma–Prozessors konnte ein großes Entwurfsspektrum unter Verwendung großer Lastprogramme ausgetestet werden. Als nächster Detaillierungsschritt beim Entwurf des Rhamma–Prozessors wurde eine VHDL–Implementierung des Rhamma–Prozessors erstellt, wobei insbesondere auf Hardware–Optimierungen wert gelegt wurde, welche die in den Software–Simulationen gefundenen Engpässe vermeiden. Wesentliche Entwurfsziele waren somit, einen möglichst schnellen Kontextwechsel und eine effiziente Implementierung der Lade/Speichereinheit zu erreichen. Um einen schnellen Kontextwechsel zu erzielen, wird ein Befehlsformat verwendet, durch das ein Kontextwechsel bereits in der Befehlholestufe der Pipeline zu erkennen ist, und es wird ein Kontextwechselpuffer eingesetzt, der die Befehlsadressen früher durchgeführter Kontextwechsel speichert. Durch diese Hardware–Maßnahmen konnte der zeitliche Aufwand für den Kontextwechsel auf einen oder gar null Prozessortakte begrenzt werden. Unter diesen Voraussetzungen wurde im Detailentwurf entschieden, nicht nur bei Lade- und Speicherbefehlen, sondern bei *allen* Befehlen, die zu Wartezeiten in der Pipeline der Ausführungseinheit führen können, einen Kontextwechsel zu implementieren. Dies schließt Sprung- und Kontext–Management–Befehle ein.

Entwicklungswerkzeuge wie das Synopsys–System erlaubten, einen in VHDL–Code beschriebenen Entwurf durch Simulationen zu validieren und mittels einer automatischen Synthese die Schaltung auf eine ASIC- oder FPGA–Technologie abzubilden. Es wurden technologische Grenzen der Implementierung gefunden und die Kosten der Optimierungen (gemessen in Chipfläche) mit der zu erwartenden Leistungssteigerung in Beziehung gesetzt. Der Pipeline–Entwurf und einige Ergebnisse des Hardware–Entwurfs werden in der vorliegenden Arbeit vorgestellt.

Im nächsten Abschnitt wird der Rhamma–Prozessor beschrieben. Danach werden die Besonderheiten der Befehlholestufe der mehrfädigen Pipeline und die Implementierung der obigen Hardware–Maßnahmen dargestellt. Im vierten Abschnitt werden die Ergebnisse der Hardware–Synthese vorgestellt.

2. Der Rhamma–Prozessor

Die Architektur des Rhamma–Prozessors (siehe Abb. 1) ist charakterisiert durch die Entkopplung von Ausführungseinheit und Lade-/Speichereinheit. Diese nebenläufig zueinander arbeitenden internen Einheiten sind durch FIFO–Puffer für sogenannte Fortsetzungen verbunden. Eine Fortsetzung setzt sich aus einer Kontextkennung und einem Befehlszähler zusammen. Um einen schnellen Kontextwechsel zu ermöglichen, sind auf dem Prozessor mehrere Registersätze vorhanden, die verschiedenen Kontrollfäden zugeordnet sind.

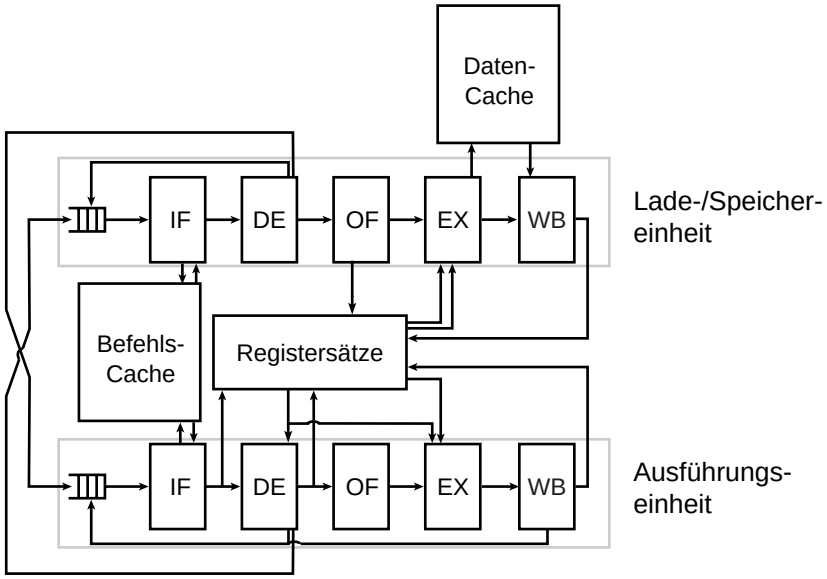

Abb. 1: Blockschaltbild des Rhamma–Prozessors

Die Ausführungseinheit lädt solange Befehle eines Kontrollfadens, bis sie auf einen Lade-, Speicher-, Sprung- oder Kontext–Management–Befehl trifft. Dann übergibt sie im Falle eines Lade-/Speicherbefehls die Fortsetzung des Kontrollfadens an die Lade-/Speichereinheit, entnimmt dem FIFO–Puffer eine weitere Fortsetzung und wechselt den Kontrollfaden durch Umschalten auf den durch die Kontextkennung der Fortsetzung angegebenen Registersatz. Die Lade-/Speichereinheit lädt parallel zur Arbeit der Ausführungseinheit Daten eines anderen Kontrollfadens in dessen Registersatz. Das Ende einer Lade- oder Speicheroperation wird durch den Rücklauf von Lade-/Speicherbestätigungen geprüft. Eine genauere Beschreibung der Prozessorarchitektur findet sich in [15].

Im Falle eines Sprung- oder Kontext–Management–Befehls wird ebenfalls ein Kontextwechsel durchgeführt. Der Befehl selbst wird noch von der Ausführungseinheit ausgeführt. Am Ende der Pipeline wird die Fortsetzung in den FIFO–Puffer der Ausführungseinheit geschrieben.

Die Ausführungs- und die Lade-/Speichereinheit des Rhamma–Prozessors wurden durch eine fünfstufige Befehls–Pipeline mit den üblichen Result–Forwarding–Techniken implementiert (Befehlhole-, Decodier-, Operandenhole-, Ausführungs- und Rückschreibestufe). Über ein Scoreboard wird die Verfügbarkeit der Operanden erkannt, bevor in der Ausführungsstufe die Berechnung durchgeführt und schließlich in der Rückschreibstufe das Resultat in das Zielregister geschrieben wird.

3. Hardware–Maßnahmen für einen schnellen Kontextwechsel

3.1 Die Befehlholestufe der Ausführungseinheit

In der Befehlholestufe wird der nächste Befehl in die Befehls–Pipeline eingefüttert. Dazu muß die Befehlsadresse an den Befehls–Cache angelegt werden. Die Adresse kann einerseits dadurch erzeugt werden, daß der nächste Befehl des aktuellen Kontrollfadens geladen wird, d.h. der Befehlszähler des Kontrollfadens wird erhöht, und andererseits, daß der Befehlszähler bei einem Kontextwechsel zusammen mit der Kontextkennung des als nächstes auszuführenden Kontrollfadens aus dem FIFO–Speicher geladen wird. Das grundlegende Schaltbild für die Befehlholestufe wird in Abb. 2 wiedergegeben. Neben dem Befehlszähler liefert die Befehlholestufe die Kontextkennung, welche den Registersatz festlegt, auf dem gearbeitet werden soll.

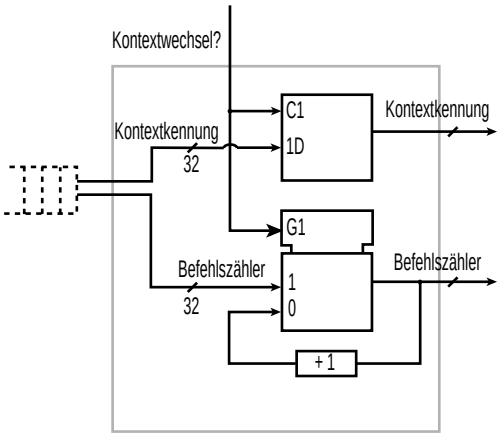

Abb. 2: Grundschaltbild der Befehlholestufe (IF) in der Ausführungseinheit

Zusätzlich zu den Bussen für die Kontextkennung und den Befehlszähler des nächsten Kontrollfadens wird an die Befehlholestufe ein Steuersignal angelegt, das den anderen Pipeline–Stufen ermöglicht, einen Kontextwechsel zu erzwingen. Ist das Kontextwechselsignal auf Eins gesetzt, so wird die Fortsetzung aus dem FIFO–Speicher übernommen und ein Kontextwechsel durchgeführt. Andernfalls gilt die Kontextkennung, die in einem transparenten Register gespeichert ist, und der neue Befehlszähler wird aus dem alten Befehlszähler berechnet.

Um Leerlauf in der Befehls–Pipeline zu verhindern, erzwingen alle Befehle, die eine Verzögerung der Befehls–Pipeline bewirken können, einen Kontextwechsel. Diese sind neben den Lade-/Speicherbefehlen auch die Sprung- und die Kontext–Management–Befehle. Diese Entwurfsentscheidung vermeidet aufwendige Hardware–Maßnahmen, die ansonsten notwendig sind, wenn in der Befehlsfolge direkt hinter einem bedingten Sprungbefehl ein Lade- oder Speicherbefehl steht, der im Falle eines Sprungs fälschlicherweise zu einem Kontextwechsel führt. Außerdem kann durch diese Maßnahme auf die Implementierung einer spekulativen Sprungauswertung und deren Rückrollmechanismus im Falle fehlgeschlagener Spekulation verzichtet werden. Dadurch werden Leerzyklen in der Pipeline–Verarbeitung vermieden. Bei der derzeitigen Implementierung wird auch nach dem Laden unbedingter Sprungbefehle ein Kontextwechsel durchgeführt. (Dies ist jedoch bei einer Erweiterung der Befehlholestufe um einen Sprungzieladreß–Cache nicht mehr nötig.) Die Methode eignet sich ebenso zur Hand-

habung von Befehlen, die Unterbrechungen auslösen können. Es ist dadurch nicht nötig, den Zustand der Befehls-Pipeline einzufrieren und nach Beendigung der Unterbrechung wieder herzustellen.

3.2 Schneller Kontextwechsel durch Prädecodierung

Da die Ausführungseinheit des Rhamma–Prozessors den Kontext nach dem Laden von Lade-/Speicher-, Sprung- oder Kontext–Management–Befehlen immer wechselt, ist es möglich, einen Kontextwechsel durch geschickte Codierung der Befehle noch vor dem eigentlichen Decodieren des Befehls zu erkennen. Das Befehlsformat und die Opcode–Codierung wurden so gewählt, daß genau die Befehle, die in der Ausführungseinheit einen Kontextwechsel bewirken, das Bit 31 des Befehlsworts gesetzt haben. Somit kann der Wert des Bits am Ausgang der Befehlholestufe als neues Steuersignal für die Befehlholestufe verwendet werden. Dies ist in Abb. 3 dargestellt. Ist der Wert des höchstwertigen Bits Eins, dann wird ein Kontextwechsel durchgeführt. Dazu werden im Falle einer Lade-/Speicheroperation der Befehlszähler und die Kontextkennung des in Bearbeitung befindlichen Kontrollfadens in den FIFO–Puffer der Lade-/Speichereinheit geschrieben und die nächste Fortsetzung aus dem FIFO–Puffer der Ausführungseinheit entnommen.

Obwohl ein Sprung oder Kontext–Management–Befehl einen Kontextwechsel auslöst, muß der Befehl von der Ausführungseinheit ausgeführt werden. Dabei wird die Fortsetzung an die folgenden Stufen der Pipeline weitergegeben. Die Fortsetzung wird erst 3 Takte später (in der Ausführungsstufe) in den FIFO–Puffer der Ausführungseinheit eingetragen. Es bleiben alle Pipeline–Stufen gefüllt, und kein Takt geht verloren.

Im Falle von Lade- oder Speicherbefehlen entsteht mit dieser Technik in der Ausführungseinheit ein Leerlauf von einem Takt für jede Folge von solchen Befehlen, weil nur der erste Lade- oder Speicherbefehl einer Folge von Lade-/Speicherbefehlen einen Kontextwechsel in der Ausführungseinheit auslöst. Aufeinanderfolgende Lade-/Speicherbefehle werden in der Lade-/Speichereinheit ausgeführt.

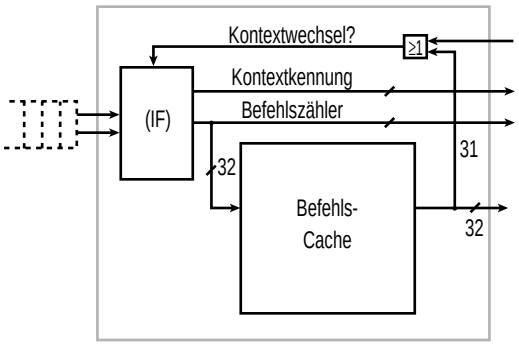

Abb. 3: Die Befehlholestufe der Ausführungseinheit mit Prädecodierung

3.3 Der Kontextwechselpuffer

Folgen von Lade-/Speicherbefehlen verursachen jeweils mit dem ersten dieser Befehle einen Kontextwechsel mit einem Leerlauftakt in der Ausführungseinheit. Um diesen Verlust bei mehrfacher Ausführung der gleichen Befehlsfolge zu vermeiden, wurde ein sogenannter *Kontextwechselpuffer* entwickelt. Dieser besteht aus einer Hardware–Tabelle, in der die Adressen von zuvor ausgeführten Lade-/Speicherbefehlen gesammelt werden, welche Kontextwechsel ausgelöst haben. Bevor nun ein Befehlswort aus dem Befehls–Cache gelesen wird, wird überprüft, ob sich die Befehlsadresse im Kontextwechselpuffer befindet. Ist dies der Fall, so wird der Kontext gewechselt und bereits mit dem nächsten Takt ein Befehl eines anderen Kontrollfadens geladen. Der Leertakt entfällt. Wird die Befehlsadresse nicht im Kontextwechselpuffer gefunden, so wird der nächste Befehl des augenblicklichen Kontrollfadens geladen und an die Decodierstufe übergeben. Sollte dabei die Befehlholestufe mittels der Prädecodierung einen Lade-/Speicherbefehl erkennen, so ergibt sich ein Leertakt durch den geladenen aber nicht in der Ausführungseinheit ausführbaren Befehl. Die Befehlsadresse wird dann in den Kontextwechselpuffer übernommen.

Der Kontextwechselpuffer ist ähnlich wie ein direkt abgebildeter (direct-mapped) Cache organisiert. Die angelegte Befehlsadresse wird in einen Index–Teil und einen Tag–Teil unterteilt. Der Index–Teil bestimmt die Zeile, in welcher der Tabelleneintrag stehen muß. Da unterschiedliche Befehlsadressen auf die gleiche Tabellenzeile abgebildet werden können, müssen beim Eintragen einer Befehlsadresse in der Tabelle die restlichen Bits der Befehlsadresse (der Tag–Teil) vermerkt werden. Der Rhamma–Prozessor läßt jede Adresse als Befehlsadresse zu, deshalb gibt ein zusätzliches Gültigkeitsbit pro Tabelleneintrag an, ob der Eintrag schon mit einer Adresse beschrieben wurde oder nicht.

Bei der Überprüfung einer Befehlsadresse wird nun zuerst mit Hilfe des Index die Zeile der Tabelle ausgewählt und danach der Tag–Eintrag mit dem Tag der Befehlsadresse verglichen. Danach wird das Ergebnis mit dem Gültigkeitsbit verknüpft, so daß genau dann eine Eins am Ausgang der Schaltung anliegt, wenn die Befehlsadresse im Kontextwechselpuffer vermerkt ist. Andere Organisationsstrukturen, wie beispielsweise eine mehrfach satzassoziative Cache–Organisation, wurden bei der Implementierung wegen der höheren Komplexität verworfen.